计算机组织与体系结构实习Lab 2

RISC-V CPU模拟器设计与实现

何昊 1600012742

一、实验环境

1.1 实验环境的安装与配置

首先，必须搭建RISC-V相关的编译、运行和测试环境。简便起见，本次实验全部基于RISC-V 64I指令集，为了配置环境，执行了如下步骤。

1. 从GitHub上下载了`riscv-tools`，从中针对Linux平台配置，编译和安装了`riscv-gnu-toolchain`。
2. 为了使用官方模拟器作为参照，从GitHub上下载、编译和安装了`riscv-qemu`。

需要特别注意的是，在编译`riscv-gnu-toolchain`时，必须指定工具链和C语言标准库所使用的指令集为RV64I，否则在编译的时候编译器会使用RV64C、RV64D等扩展指令集。即使设置编译器编译时只使用`RV64I`指令集，编译器也会链接进使用扩展指令集的标准库函数。因此，为了获得只使用RV64I标准指令集的ELF程序，必须在`riscv-gnu-toolchain`中采用如下选项重新编译

```
mkdir build; cd build
../configure --with-arch=rv64i --prefix=/path/to/riscv64i
make -j$(nproc)
```

在编译时，使用`-march=rv64i`让编译器针对RV64I标准指令集生成ELF程序。

```
riscv64-unknown-elf-gcc -march=rv64i test/arithmetic.c test/lib.c -o riscv-elf/arithmetic.riscv
```

1.2 使用的测试程序和测试方法

对一个体系结构模拟器进行测试有一定难度，主要是由于指令数众多、代码庞大、从而对模拟器代码进行100%覆盖率的测试比较困难。因此，为了便于测试，本模拟器使用了一组由简单到复杂的测试程序，并且实现了单步调试和打印CPU状态的接口。此外，为了便于进行调试和性能分析，还实现了记录执行历史的模块，在程序出错时可以获得完整的指令执行历史和内存快照，便于对出错进行分析。

为了对RISC-V模拟器进行测试，编写了如下程序（见`test/`文件夹）。比较复杂的是快速排序、矩阵乘法和求Ackermann函数三个。其中，快速排序和矩阵乘法涉及比较多的指令和数据，求解Ackermann函数涉及非常深的递归调用。

```
lib.c             # 自定义的系统调用实现
helloworld.c      # 最简单的程序
test_arithmetic.c # 对运算指令的测试
test_branch.c     # 对基本分支的测试
test_syscall.c    # 对系统调用的测试
quicksort.c       # 快速排序
matrixmulti.c     # 矩阵乘法
ackermann.c       # 求解Ackermann函数
```

所有程序编译后得到的二进制程序和反编译得到的汇编代码均保存在`riscv-elfs/`文件夹中。

二、设计概述

2.1 开发环境

我测试的模拟器运行环境为Mac OS X，使用的编程语言为C++ 11，构建环境为CMake，编译器为`Apple Clang 10.0.0`，编译使用的Flag为`-O2 -Wall`。开发使用的工具为VS Code。不过，模拟器代码尽量避免了使用标准库以外的平台相关功能，所以应该也能在其他平台和编译器上编译运行。

2.2 设计考量

首先，模拟器的运行必须是健壮的。具体地说，必须能够处理各种非法输入，包括不正常的访存，不正常的ELF文件，非法指令，非法的访存地址等等。编写细致全面的错误处理不仅有助于锻炼系统编程能力，也有助于在早期发现细微的程序错误。

其次，模拟器的实现必须简单、易于理解和易于调试。此模拟器是一个课程项目级别的模拟器，允许的实现时间有限，因此代码实现必须简单，调试系统必须完备，从而尽可能地减少编写程序和调试程序所需要的时间。

此外，模拟器实现的主要目的是能够被用于简单性能评测，因此必须能够尽可能贴近流水线硬件，并可以扩展出分支预测和缓存模拟等各种功能，便于在真正的程序上实验和评测流水线的性能，以及各种分支预测和缓存模拟策略。

本次模拟器的实现并不是要做一个成熟可用的工业级体系结构模拟器，也就是说，本次模拟器的实现并不注重性能和功能的全面性。在性能上，对于极端复杂和庞大的程序，模拟器的程序会执行缓慢，也有可能会消耗过多内存，对于模拟器本身的性能优化不在本实验的范围内。在功能上，为了实现简单，本模拟器使用自定义的系统调用，而不是符合Linux的系统调用，因此，此模拟器只能运行专门为此编译的RISC-V程序（编写方法参见`test/`文件夹）。

2.3 编译与运行

编译方法与一个典型的CMake项目一样，在编译之前必须先安装CMake。在Linux或者Mac OS X系统上可以采用如下命令

```
mkdir build
cd build
cmake ..
make
```

编译会得到可执行程序`Simulator`。该模拟器是一个命令行程序，在命令行上的执行方式是

```
./Simulator riscv-elf-file-name [-v] [-s] [-d] [-b param]
Parameters:
        [-v] verbose output
        [-s] single step
        [-d] dump memory and register trace to dump.txt
        [-b param] branch perdiction strategy, accepted param AT, NT, BTFNT
```

其中`riscv-elf-file-name`对应可执行的RISC-V ELF文件，比如`riscv-elf/`文件夹下的所有`*.riscv`文件。一个典型的运行流程和输出如下

```
hehaodeMacBook-Pro:build hehao$ ./Simulator ../riscv-elf/ackermann.riscv
Ackermann(0,0) = 1
Ackermann(0,1) = 2
Ackermann(0,2) = 3
Ackermann(0,3) = 4
Ackermann(0,4) = 5
Ackermann(1,0) = 2
Ackermann(1,1) = 3
Ackermann(1,2) = 4
Ackermann(1,3) = 5
Ackermann(1,4) = 6
Ackermann(2,0) = 3
Ackermann(2,1) = 5
Ackermann(2,2) = 7
Ackermann(2,3) = 9
Ackermann(2,4) = 11
Ackermann(3,0) = 5
Ackermann(3,1) = 13
Ackermann(3,2) = 29
Ackermann(3,3) = 61
Ackermann(3,4) = 125
Program exit from an exit() system call
------------ STATISTICS -----------
Number of Instructions: 430754
Number of Cycles: 574548
Avg Cycles per Instrcution: 1.3338
Branch Perdiction Accuacy: 0.5045 (Strategy: Always Not Taken)
Number of Control Hazards: 48010
Number of Data Hazards: 279916
Number of Memory Hazards: 47774
---------------------------------
```

在默认的设置下，一开始会首先打印执行的程序的输出，然后会输出一组关于CPU执行情况的统计数据。

如果要进行单步调试的话，可以类似如下使用`-s`和`-v`参数

```
./Simulator ../riscv-elf/ackermann.riscv -s -v
```

得到的输出如下

```
hehaodeMacBook-Pro:build hehao$ ./Simulator ../riscv-elf/ackermann.riscv -s -v
==========ELF Information==========
Type: ELF64
Encoding: Little Endian
ISA: RISC-V(0xf3)
Number of Sections: 19
ID      Name            Address Size
[0]                     0x0     0
[1]     .text           0x100b0 3668
[2]     .rodata         0x10f08 29
[3]     .eh_frame       0x10f28 4
[4]     .init_array     0x11000 8
[5]     .fini_array     0x11008 8
[6]     .data           0x11010 1864
[7]     .sdata          0x11758 24
[8]     .sbss           0x11770 8
[9]     .bss            0x11778 72
[10]    .comment        0x0     26
[11]    .debug_aranges  0x0     48
[12]    .debug_info     0x0     46
[13]    .debug_abbrev   0x0     20
[14]    .debug_line     0x0     222
[15]    .debug_str      0x0     267
[16]    .symtab         0x0     2616
[17]    .strtab         0x0     913
[18]    .shstrtab       0x0     172
Number of Segments: 2
ID      Flags   Address FSize   MSize
[0]     0x5     0x10000 3884    3884
[1]     0x6     0x11000 1904    1984
==================================
Memory Pages:
0x0-0x400000:
  0x10000-0x11000
  0x11000-0x12000
Fetched instruction 0x00002197 at address 0x100b0
Decode: Bubble
Execute: Bubble
Memory Access: Bubble
WriteBack: Bubble
------------ CPU STATE ------------
PC: 0x100b4
zero: 0x00000000(0) ra: 0x00000000(0) sp: 0x80000000(2147483648) gp: 0x00000000(0)
tp: 0x00000000(0) t0: 0x00000000(0) t1: 0x00000000(0) t2: 0x00000000(0)
s0: 0x00000000(0) s1: 0x00000000(0) a0: 0x00000000(0) a1: 0x00000000(0)
a2: 0x00000000(0) a3: 0x00000000(0) a4: 0x00000000(0) a5: 0x00000000(0)
a6: 0x00000000(0) a7: 0x00000000(0) s2: 0x00000000(0) s3: 0x00000000(0)
s4: 0x00000000(0) s5: 0x00000000(0) s6: 0x00000000(0) s7: 0x00000000(0)
s8: 0x00000000(0) s9: 0x00000000(0) s10: 0x00000000(0) s11: 0x00000000(0)
```

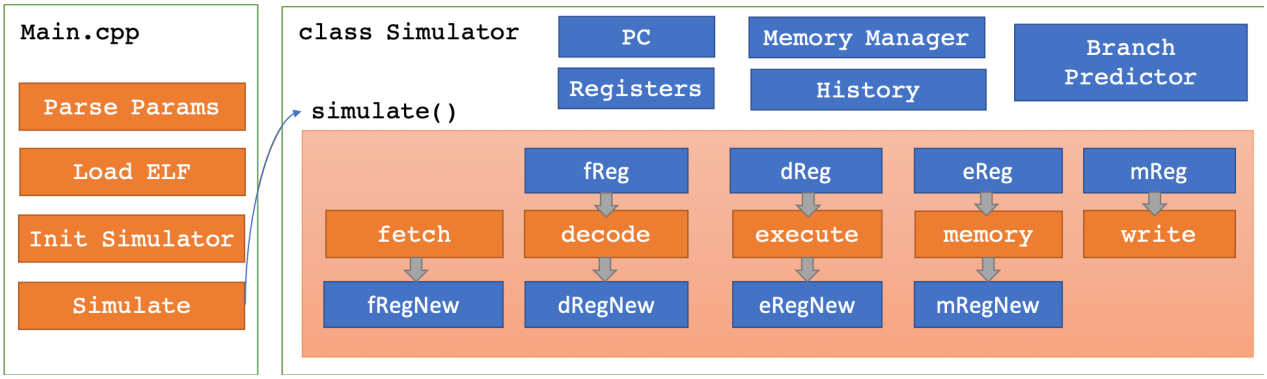
```
t3: 0x00000000(0) t4: 0x00000000(0) t5: 0x00000000(0) t6: 0x00000000(0)
----------------------------------
Type d to dump memory in dump.txt, press ENTER to continue:
```

在单步调试中，可以输入d来保存内存快照，使用ENTER前进到下一条指令。命令行显示的信息包括ELF信息、流水线状态和CPU寄存器状态。

此外，可以使用-b参数指定不同的分支预测策略，例如

```
./Simulator ../riscv-elf/ackermann.riscv -b AT
./Simulator ../riscv-elf/ackermann.riscv -b NT
./Simulator ../riscv-elf/ackermann.riscv -b BTFNT
```

2.4 代码架构



三、具体设计和实现

3.1 可执行文件的装载、初始化和存储接口

3.2 指令语义的解析和控制信号的处理（如果有）

3.3 系统调用和库函数接口的处理

3.4 性能计数相关模块的处理

3.5 调试接口和其它接口等

3.6 分支预测模块的实现

3.7 实现中遇到的坑

在整个实现中，我在第一阶段的单周期指令级模拟的实现并没有遇到什么问题，但是流水线相关的模拟中，遇到了好几个相当微妙的错误。

1. 一个根本的困难在于我们对流水线的模拟程序本质上还是线性执行的，并不能像硬件那样多阶段并行执行。因此，必须非常小心地设计五个阶段的代码的执行流和对数据结构的访问，才能模拟出硬件的效果。
2. 当多个阶段发现数据冒险并向前转发数据时，必须优先传送更新的数据。在模拟器中，由于相关

阶段的执行顺序是执行->访存->写回，因此会存在前面的阶段向前转发的数据被后面的阶段的旧数据覆盖的可能。对于这种情况，模拟器中必须加以特别的判定。

3. 分支预测模块应当在解码阶段根据预测结果修改PC的值，但是，如果这个跳转指令是被错误取进来，并且应该在之后被Bubble的话怎么办？必须想办法恢复被修改的PC值，或者延迟写入预测的PC值。
4. 也是由于代码是顺序执行的，因此当执行阶段发现访存指令，而解码阶段的指令依赖访存数据并导致内存冒险时，必须非常小心地设计整个执行过程和数据访问流程，才能模拟出正确的结果。
5. 用于系统调用的`ecall`指令也会导致数据冒险！并且产生数据冒险的条目，取决于这个系统调用的参数数量和其对应的寄存器！当前的系统调用会依赖的寄存器有`a0`和`a7`两个，因此刚好能作为`op1`和`op2`塞入流水线，但是如果系统调用需要的参数更多，实现将会变得更为复杂。
6. `zero`寄存器是一个相当独特的存在，理论上他任何时候值应该都是0，所以进行数据转发的时候必须处处特判零寄存器，如果向零寄存器里的值进行数据转发就会导致非常难以发现的错误。

四、功能测试和性能评测

4.1 运行测试程序，给出动态执行的指令数。（共5个定点程序）

```
Program exit from an exit system call
------------ STATISTICS -----------
Total Number of Instructions Executed: 910
-----------------------------------
Program exit from an exit system call
------------ STATISTICS -----------
Total Number of Instructions Executed: 935
-----------------------------------
Program exit from an exit system call
------------ STATISTICS -----------
Total Number of Instructions Executed: 1129
-----------------------------------
Program exit from an exit system call
------------ STATISTICS -----------
Total Number of Instructions Executed: 19484
-----------------------------------
Program exit from an exit system call
------------ STATISTICS -----------
Total Number of Instructions Executed: 920
-----------------------------------
```

4.2 运行测试程序，给出执行周期数，并计算平均CPI。（共5个定点程序）

4.3 请你模拟的流水线处理器中因不同类型的冒险而发生的停顿进行统计，并打印数据和分析。（共5个测试程序）

4.4 分支预测模块评测

五、其它需要说明的内容

5.1 额外的功能或性能特性，更优化的设计等

5.2 意见和建议

1. RISC-V工具链文档缺失
2. 计算机体系结构正课教的是MIPS，不知道为什么Lab却要做RISC-V，增加了学习成本和完成Lab的时间